What is Biopython?

The Biopython Project is an international association of developers of freely available Python (http://www.python.org) tools for computational molecular biology. The Biopython web site (http://www.biopython.org) provides an online resource for modules, scripts, and web links for developers of Python-based software for bioinformatics use and research. Basically, the goal of Biopython is to make it as easy as possible to use Python for bioinformatics by creating high-quality, reusable modules and classes. Biopython features include parsers for various Bioinformatics file formats (BLAST, Clustalw, FASTA, Genbank,...), access to online services (NCBI, Expasy,...), interfaces to common and not-so-common programs (Clustalw, DSSP, MSMS...), a standard sequence class, various clustering modules, a KD tree data structure etc. and even documentation.

The main Biopython releases have lots of functionality, including:

- **The ability to parse bioinformatics files into Python utilizable data structures, including support for the following formats:**
  - Blast output – both from standalone and WWW Blast
  - Clustalw
  - FASTA
  - GenBank
  - PubMed and Medline
  - ExPASy files, like Enzyme and Prosite
  - SCOP, including 'dom' and 'lin' files
  - UniGene
  - SwissProt
- **Files in the supported formats can be iterated over record by record or indexed and accessed via a Dictionary interface.**
- Code to deal with popular on-line bioinformatics destinations such as:
  - NCBI – Blast, Entrez and PubMed services
  - ExPASy – Swiss-Prot and Prosite entries, as well as Prosite searches
- Interfaces to common bioinformatics programs such as:
  - Standalone Blast from NCBI
  - Clustalw alignment program
  - EMBOSS command line tools
- **A standard sequence class that deals with sequences, ids on sequences, and sequence features.**

- **Tools for performing common operations on sequences, such as translation, transcription and weight calculations.**
- Code to perform classification of data using k Nearest Neighbors, Naive Bayes or Support Vector Machines.
- **Code for dealing with alignments, including a standard way to create and deal with substitution matrices.**
- GUI-based programs to do basic sequence manipulations, translations, BLASTing, etc.
- Integration with BioSQL, a sequence database schema also supported by the BioPerl and BioJava projects.

Biopython Exercises

1. *Write a function in python for calculating the GC content of a sequence using biopython.*
2. *Write a program to calculate the reverse complement of a sequence using biopython.*
3. *Write a program to translate a DNA sequence to a protein sequence using biopython.*
4. *Write a program to parse a .fasta file using biopython.*
5. *Write a program to parse a genbank file using biopython.*
6. *Write a program to reverse complement a sequence using biopython.*
7. *Write a program to parse a .xml file using biopython.*
8. *Write a program to perform a sequence alignment using biopython.*
9. *Write a program to read a sequence alignment using biopython.*

Working with sequences

```
from Bio.Seq import Seq

my_seq = Seq("AGTACACTGGTA")
print("Sequence: ", my_seq)

my_seq_complement = my_seq.complement()
print("Complementary Sequence: ", my_seq_complement)

my_seq_reverse_comp = my_seq.reverse_complement()
print("Reverse Complement: ", my_seq_reverse_comp)

my_seq_trb = my_seq.transcribe()
print("Transcription: ", my_seq_trb)

my_seq_trl = my_seq.translate()
print("Translation: ", my_seq_trl)
```

```
gc = GC(my_seq)
print("GC Content: ", gc)

mw = molecular_weight(my_seq)
print("Molecular Weight: ", mw)
```

Parsing sequence file formats, Sequence Input/Output
Simple FASTA parsing example

```
from Bio import SeqIO

for seq_record in SeqIO.parse("test.fasta", "fasta"):
  print("ID: ", seq_record.id)
  print("Seq: ", seq_record.seq)
  print("Seq Length: ", len(seq_record))
```

Simple GenBank parsing example

```
from Bio import SeqIO
for seq_record in SeqIO.parse("genbankfile.gbk", "genbank"):
  print("ID: ", seq_record.id)
  print("Seq: ", seq_record.seq)
  print("Seq Length: ", len(seq_record))
```

Sequences and Alphabets

You should specify the alphabet explicitly when creating your sequence objects as protein/DNA alphabet object:

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC

my_dna_seq = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)
print("DNA Seq: ", my_dna_seq)
print("Alphabet: ", my_dna_seq.alphabet)

my_prot_seq = Seq("AGTAMCACTGGT", IUPAC.protein)
print("Protein Seq: ", my_prot_seq)
print("Alphabet: ", my_prot_seq.alphabet)
```

Sequences act like strings

In many ways Seq objects can be worked with as normal Python strings.

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC

my_seq = Seq("GATCG", IUPAC.unambiguous_dna)
for index, letter in enumerate(my_seq):
   print("%i %s" % (index, letter))

print(len(my_seq))

print(my_seq[0]) #first letter
print(my_seq[2]) #third letter
print(my_seq[-1]) #last letter

print(my_seq.count('A'))
print(my_seq.count("AA"))

# Calculating GC content in usual way
my_seq = Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPAC.unambiguous_dna)
gc = 100 * float(my_seq.count("G") + my_seq.count("C")) / len(my_seq)
print("GC = ", gc)

# Slicing asequence
my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
print("A Slice of the Seq: ", my_seq[4:12])

# Changing case
print(my_seq.upper())
print(my_seq.lower())
```

Writing Sequence Files

```
from Bio.Seq import Seq
from Bio import SeqIO
from Bio.SeqRecord import SeqRecord
from Bio.Alphabet import generic_protein

rec1 =
SeqRecord(Seq("MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALF
GD",  generic_protein), id="gi|14150838|gb|AAK54648.1|AF376133_1",
```

```
        description="chalcone synthase [Cucumis sativus]")
rec2 =
SeqRecord(Seq("YPDYYFRITNREHKAELKEKFQRMCDKSMIKKRYMYLTEEILKENPSMCEYMAPSLDA
RQ", generic_protein), id="gi|13919613|gb|AAK33142.1|",
        description="chalcone synthase [Fragaria vesca subsp. bracteata]")

rec3 =
SeqRecord(Seq("MVTVEEFRRAQCAEGPATVMAIGTATPSNCVDQSTYPDYYFRITNSEHKVELKEKFKR
MC", generic_protein), id="gi|13925890|gb|AAK49457.1|",
        description="chalcone synthase [Nicotiana tabacum]")

my_records = [rec1, rec2, rec3]
# write to a FASTA format file
SeqIO.write(my_records, "my_example.fasta", "fasta")
```

Converting between sequence file formats

```
from Bio import SeqIO
records = SeqIO.parse("orchid.gbk", "genbank")
SeqIO.write(records, "my_example.fasta", "fasta")
SeqIO.convert("orchid.gbk", "genbank", "another_example.fasta", "fasta")
```

Parsing or Reading Sequence Alignments

```
from Bio import AlignIO
alignment = AlignIO.read("my_alignment.fasta", "fasta")
print(alignment)
# or you can see record by record
i = 1
for record in alignment:
  print(i, record)
  i += 1
```

Biopython's pairwise2

```
from Bio import SeqIO
from Bio import pairwise2

seq1 = "MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALFGD"
seq2 = "YPDYYFRITNREHKAELKEKFQRMCDKSMIKKRYMYLTEEILKENPSMCEYMAPSLDARQ"
alignments = pairwise2.align.globalxx(seq1, seq2)
print("Length: ", len(alignments))
print(alignments[0])
```

Phylogenetics with Bio.Phylo
Read the newick file
Create a simple Newick file named simple.dnd using a text editor and write in the file:
(((A,B),(C,D)),(E,F,G));

```
from Bio import Phylo
tree = Phylo.read("simple.dnd", "newick")
print(tree)
# Or you can print a tree
Phylo.draw_ascii(tree)
```

Sequence motif analysis, creating a motif from instances

```
from Bio.Seq import Seq
from Bio import motifs
instances =
[Seq("TACAA"),Seq("TACGC"),Seq("TACAC"),Seq("TACCC"),Seq("AACCC"),Seq("AATGC"),Seq("AAT
GC")]
# create a Motif object
m = motifs.create(instances)
print(m)
print("Length: ", len(m))

#the counts of each nucleotide at each position
print(m.counts)
print(m.counts['A'])
print(m.counts['T', 0])
print(m.counts[:, 3])
print(m.alphabet)
print(m.alphabet.letters)
print("Concensus: ", m.consensus)
print("Anti Concensus: ", m.anticonsensus)
# Sequence logo
m.weblogo("mymotif.png")
```